CHAPTER 20
Performance

Lisp programmers know the value of everything and the cost of nothing.

—Alan Perlis

Introduction

This chapter examines methods for discovering why a Lisp program might be underperforming. I take it that you already know all about the perils of *premature optimization*: we can take it that the program is believed to be working correctly in all aspects other than performance. The experimental techniques discussed here are great for locating bottlenecks in large applications. They apply equally whether you wrote the code yourself or have never seen it before. Therefore I will assume that you do not necessarily know anything about the internals of the program under investigation.

Typically, once you know where your bottlenecks are, the correct response is to recode them for *efficiency*.

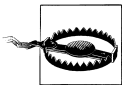
- Think carefully about algorithms.
- Use appropriate data structures (see "Data" on page 15).
- Avoid expensive calculations; cache intermediate results.
- Don't expect the application to discover facts which you could have told it in advance.

Sometimes you'll get to the end of this process and find that you're still running too slow. This is the point at which you should turn to Lisp's built-in support for optimization; we'll look at this at the end of the chapter.

As a driving example, we'll return to the ch-image library of Chapter 17 and consider the time it spends loading a largish JPEG image. We'll continue to work with Clozure Common Lisp under SLIME on FreeBSD.

```
CL-USER> (time (ch-image:read-image-file "~/chapter-20/test.jpg"))
(CH-IMAGE:READ-IMAGE-FILE "~/chapter-20/test.jpg") took 4,102 milliseconds
...
#<CH-IMAGE:ARGB-8888-IMAGE #x34EAFBB6>
CL-USER>
```

Our mission then is to find out why ch-image takes over four seconds to load a 300k image file (on this particular machine). Is this time reasonable? What can we do to improve on it? Let's find out.



I really don't want to suggest that I have any complaints whatsoever about the performance of ch-image; it just happens to make a good example which fits in with earlier chapters. In practice the question is always: are you meeting your requirements? If your application takes too long to run then something needs to be done to make it faster. If execution times are satisfactory then inefficiency is not a problem and you should not waste effort on performance. If your code doesn't work correctly then your attention should be elsewhere.

Profiling Tools

Needless to say, before you start looking at performance issues you will ensure that all of your code is compiled (see Chapter 12). Interpreted code has one or two interesting benefits but speed isn't one of them.

We know that a certain operation takes four seconds but we don't know why. We'd like to break the operation down into components, and break these components down, and so on until—maybe—we find a culprit. We can `cl:time` the operation as a whole but can't naively apply this to every function call it makes.

Exercise

Why not?

There is however a powerful tool which can manage this for us, namely a *code profiler*. Although this is not part of Common Lisp, most implementations ship with one. The details differ from place to place but they all work along similar principles and are used in more or less the same way; you should have little difficulty transferring the specifics of this chapter to whatever system you're using.

Unfortunately when we turn to Clozure CL we find that although it ships two profilers, one only works on Mac OS and the other on Linux. On FreeBSD we'll have to look elsewhere for help.

If your implementation provides a profiler then you should use that in preference to anything else. It will load and run without a fuss and it might be able to monitor more "system" functions than a portable profiler could.

We'll be working here with a public domain library called the *Metering System*. You'll find a download link for this here:

http://cl-user.net/asp/libs/Metering

The Metering System was written in the 1980s and appears not to have been maintained at all since 1994. The dates indicate that it might not quite conform to the ANSI standard. Indeed it predates CLOS. Also there are no developer mailing lists to which you can turn for help if you get stuck. On the other hand we'll see that its age hasn't detracted from its usefulness, so please don't be put off.

Building the Profiler

The Metering System consists of a single source file, *metering.cl*. (You will encounter the *.cl* filetype for Lisp files now and then.) There are some obstacles to using the file as-is: it's not quite ANSI Lisp, and it needs to be configured for the implementation we're using.

SLIME for both Clozure CL and CLISP automatically loads a copy of *metering* in which these problems have already been fixed. (That's another good reason for using SLIME.) Don't regard the following as a wasted lesson though: occasionally as a Lisp programmer you will come across files which need this level of attention. Having the confidence to proceed with them is a valuable asset.

In the first edition of *Common Lisp the Language* (*CLtL1*), `in-package` was a *function* which took keyword arguments `:nicknames` and `:use` and which was used to create new packages on the fly. In the second edition (*CLtL2*) and subsequent ANSI specification, an incompatible change was introduced and `in-package` became the *macro* it is today. It does not take any keywords and it may only refer to pre-existing packages; to define and configure a new package you'll typically use `defpackage` (which takes `:nicknames`, `:use` and seven other keywords).

The Metering System tries to work around this using the `:cltl2` feature (see line 378):

```
#-:cltl2
(in-package "MONITOR" :nicknames '("MON"))
```

followed by package definitions appropriate to non-CLtL2 compatible Lisps (which at the time meant: CLtL1 compatible). Unfortunately, although this code would have been fine when it was first written, ANSI gave `#+:cltl2` a specific meaning: it implies

that the implementation does not conform to the standard (because it conforms instead to CLtL2). So, quite reasonably, no modern Lisp carries the `:cltl2` feature; the old-style `in-package` above will not be skipped by the reader and an error will be signaled.

We can fix this by commenting out the above form (and the `#-:cltl2`) and replacing them with:

```
(defpackage "MONITOR" (:nicknames "MON"))
(in-package "MONITOR")
```

Let's turn now to configuration for Clozure CL. Almost all of the Metering System is written in portable (if somewhat ancient) Common Lisp. Four optional implementation-dependent functions provide hooks into the underlying Lisp for tailoring it. I don't want to repeat the level of archaeological details above; suffice it to say that these functions are already defined in a form suitable for Clozure but that further feature skew renders them invisible. You can remedy this, by pushing both `:mcl` and `:ccl-2` onto `*features*`.

Unfortunately, again, the `:mcl` feature causes minor problems of its own. You'll have to comment out two forms: the `#+:mcl defpackage` which imports `provide` and `require` from the `CCL` package, and the `#+:mcl` form `(ccl:provide "monitor")`—or change this to `(provide "monitor")`.

Exercise

Volunteer to maintain the Metering System: make it conform to ANSI Common Lisp and use features appropriate to recent versions of all the implementations you think it ought to support.

The `:mcl` feature happens to work in this context. This should not be taken to imply that MCL (*Macintosh Common Lisp*) and Clozure CL are compatible, even though they are forks off the same code base. Remove this feature before compiling or loading any other libraries.

Now that we've got all this out of the way we can return to the problem in hand.

Profiling in Action

The task of a profiler is to run your code and reporting back on execution times. The exact list of features on offer will vary from case to case, but you can expect any profiler to provide at least these three basic pieces of information about each function under investigation:

Call count
: How many times was this function called?

Inclusive time
: How much time was spent either in this function or in the functions it called? In other words, how often was this function on the execution stack?

Exclusive time
: How much time was spent running the code in this function alone? Equivalently, how often was this function at the top of the stack?

Some profilers present their results in the form of a *call tree*, which can be incredibly useful. It's one thing to find out that you spent 15% of your runtime in `format`, it's quite another to know which of dozens of callers to `format` from your application was responsible for this drain. The Metering System doesn't offer this facility though, so we'll have to get along without it.

Configuring the Profiler

The basic steps to using a profiler are:

1. Tell it which functions you want it to watch ("mark them for profiling").
2. Run the code you want profiled.
3. Examine results; refine procedure and/or modify code; repeat the cycle.

Most of the fine details are library-specific; if you're using your implementation's profiler, you're going to have to read about them in product documentation. The general picture though will be similar whichever profiling library you use. (By the way, for the Metering System, library documentation takes the form of a 200-line comment block near the head of *metering.cl*.)

Profilers such as the Metering System work by redefining a specified set of functions so that they record call counts, timing, allocation, etc., in addition to performing their normal actions. Other profilers might use *statistical sampling* of the call stack in addition to (or instead of) function redefinition. In either case profiled code will run an order of magnitude slower than it would normally. Depending on your profiling library, any redefinition and associated slow-down will apply either after step 1 above, or for the duration of step 2 but no longer, or once you've commenced step 2 and until you explicitly *unprofile* things. Three other consequences of this redefinition:

- If you redefine functions yourself they might cease to be profiled unless you explicitly mark them again. (This is the case with the Metering System. It's less likely to be the case with implementation-specific profilers.)
- All this might interact badly with tracing, which also works by redefining functions. The Metering System doesn't do well here (if you `trace` a function and then mark it, untracing it will have no effect) but again you can expect better behavior from the implementations' profilers.

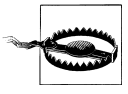
- The Metering System is not aware of CLOS. It can't profile individual methods and you can't add to or redefine the methods of a generic function while it is marked for profiling.

Exercise

Implement a basic `trace` / `untrace` facility: one function which closes over an original definition, printing out call arguments and return values, and is otherwise unobtrusive to the application; another using the same closure to restore the original definition. Don't worry about all the fine details of `cl:trace`. How straightforward is it (a) to ensure a function stays traced when you redefine it and (b) to co-operate with SLIME's implementation hooks to figure out a function's argument list?

Exercise

Delve into *metering.cl*. Redefine the functions `monitoring-encapsulate` and `monitoring-unencapsulate` so that CLOS methods are profiled too.



Don't run timing tests in an image in which you have been using a profiler. Don't run a profiler in a live server unless you really know what you're doing.

The Metering System provides a number of methods for marking functions. I take the attitude that it's better to profile too much than too little, so I tend to mark whole packages at a time. My choice is therefore to use the function

```
(monitor:monitor-all &optional (package *package*))
```

which marks every function in `package`, along with the *macro*

```
(monitor:unmonitor &rest names)
```

which—in the style of `untrace`—either unmarks the named functions or if no names are specified unmarks everything.

In our ch-image example, we really don't know in advance why image loading takes as long as it does. So it's worth groveling through the source and making a list of the packages in which ch-image and its dependencies are defined.

Exercise

Write a utility for listing the packages created during the load of the `"ch-image"` system.

In the first instance, you should steer clear of profiling implementation packages (such as `CCL` in Clozure CL). You'd like to start your investigations hoping that the underlying Lisp implementation is not at fault; also there may be problems if a profiler which wasn't shipped by your Lisp's implementors attempts to redefine system internals. (As it happens, on Clozure CL the Metering System can't monitor either `COMMON-LISP` or Clozure's internals package `CCL`.)

We're going to spend a lot of time looking at function definitions. I recommend using SLIME to locate them. Incidentally SLIME comes with a profiling interface. I didn't mention it in Chapter 18 as I am not convinced that there's much value in using it.

Exercise

Why not?

So, finally:

```
CL-USER> (dolist (package '(ch-image ch-util clem jpeg))
           (monitor:monitor-all package))
NIL
CL-USER>
```

Run the Profiler

With the Metering System, functions are redefined ready for profiling as soon as you mark them. To profile your code, simply run it.

```
CL-USER> (ch-image:read-image-file "~/chapter-20/test.jpg")
;;  This won't be quick. Go make yourself a cup of tea.
#<CH-IMAGE:ARGB-8888-IMAGE #x36686C76>
CL-USER>
```

With other profilers you may have to tell your system that you're actively profiling something: `(system:profile (my-function-call))`, or whatever. Look it up.

If you want to profile your application's response to an end user interaction (for example, mouse events in a GUI, or page requests for a web server), you should write a function which mimics the user's gesture and profile that.

If the function which you intend to feed into the profiler has arguments which are hard to recreate by hand, use `trace` or `break` or outright redefinition to capture them.

Some profilers (not this one) work by repeatedly interrupting the computation to examine the stack. This has implications on exactly what you should profile.

- There's little to be learned from profiling a thread that's waiting; so if processing the interaction involves a thread switch then profile calls within the thread which does the hard work.
- If it's all over too quickly you won't get enough readings to form an accurate picture. A good rule of thumb is to profile code which (unprofiled) would take a few seconds to run. If necessary, put the code you're interested in into a loop and profile that.

Results: Exclusive Times

Profiling is an experimental process. By the end of the afternoon you'll have filled your listener with the results of a whole load of profiler runs and if you're not careful you'll find yourself buried under a pile of numbers. For the sake of brevity I'm going to gloss over my own mess here and present just the essential findings, as if magically I'd gone straight to them.

Make careful records as you go so that your results are reproducible and you can reconstruct later precisely what state the code was in each time you ran the profiler. A good way to do this is to leave complete versions in the source files of everything you've tried, appropriately conditionalized (the first two changes we make below might be marked `#+mref-as-macro` and `#+defun`).

If you've had to activate the profiler explicitly (by wrapping your code in a call to `system:profile` or some such) then maybe it will print its results automatically. Otherwise, your documentation will supply you with a simple call to make.

In our case, the function we want is `monitor:report`. This generates more information than I want to discuss here; I've cut from the table below three columns which detail each function's total allocation (referred to there as *consing*—see "How It Works, Optional Second Pass" on page 0 in Chapter 16). Sometimes this data can be helpful, particularly if you come to suspect that performance is GC-bound. But you should start your search without any such assumptions, and that's what we're going to do here*.

```
CL-USER> (monitor:report)

                          %                     Total
Function                  Time  Calls    Sec/Call  Time
------------------------------------------------------------
CH-IMAGE:SET-ARGB-VALUES: 0.75  1920000  0.000019  36.97800
CH-IMAGE:MAP-PIXELS:      0.10        1  4.930997  4.930997
```

* Don't let the % upset you too much.

```
CH-IMAGE:SET-PIXEL:       0.09  1920000  0.000002  4.306000
JPEG::DECODE-CHUNK:       0.03        1  1.482998  1.482998
CLEM::MATRIX-VALS:        0.03  7680008  0.000000  1.237982
---------------------------------------------------------
TOTAL:                    0.99  11520010           48.935978
...
CL-USER>
```

Interpretation

The report above shows two of the three pieces of information we listed in "Profiling in Action" on page 4: call count and exclusive time. Straight away we know that three quarters of the execution time was spent in calls to just one function: `ch-image:set-argb-values`. Let's confirm this.

There's another reason for writing the following test: in order to measure success when we start making changes to the code, it can be useful to time these calls in isolation from the rest of the application.

```
CL-USER> (monitor:unmonitor)
; No value
CL-USER> (defun test (n)
           (let ((img (make-instance 'ch-image:argb-8888-image
                                     :width 1 :height 1)))
             (dotimes (i n)
               (ch-image:set-argb-values img 0 0 0 0 0))))
TEST
CL-USER> (time (test 1920000))
(TEST 1920000) took 3,256 milliseconds (3.256 seconds) to run
...
CL-USER>
```

When we take a look at `ch-image:set-argb-values`, it turns out to be a generic function and SLIME's `meta-.` offers us a choice of two methods, one specializing on `ch-image:argb-image`, the other on its direct subclass `ch-image:argb-8888-image`. How do we know which one to look at? Well, we don't at first. So we're going to have to devise a way of finding out.

Exercise

Tracing a function which is going to be called two million times is inadvisable. (Why?) Write a `:before` method for `ch-image:set-argb-values` which records the class of its first argument. If you want to try this out with the Metering System, don't forget to `(monitor:unmonitor ch-image:set-argb-values)` first. When you're done, remove your method. You need to know for certain that what you're testing is "clean"; if you get badly tangled, restart Lisp.

In this case, it turns out that the joke's on me because when we go to look at the source we find the two definitions side by side and apart from their specializers they are identical. A possible explanation for this coding style is that the compiler can depend on the types of specialized arguments thus sometimes making slot accesses faster. As CLOS guarantees run-time type checks before selecting the method, this optimization is not at the expense of *safety*: if an argument is not of the promised type the method cannot be selected.

Exercise

Remove one of the two methods (does it matter which one?) and time the test again. Did this make any improvement? (Why might it?)

Anyway here, thanks to SLIME and `M-.`, is the definition we're after along with some supporting classes and methods.

```
(in-package "CH-IMAGE")

(defclass argb-image (multichannel-image)
  ((a :accessor image-a)
   (r :accessor image-r)
   (g :accessor image-g)
   (b :accessor image-b)))

(defclass argb-8888-image (argb-image) ())

(defmethod set-argb-values ((img argb-8888-image)
                            (row fixnum) (col fixnum)
                            (a fixnum) (r fixnum) (g fixnum) (b fixnum))
  (setf (mref (image-a img) row col) a)
  (setf (mref (image-r img) row col) r)
  (setf (mref (image-g img) row col) g)
  (setf (mref (image-b img) row col) b))
```

and

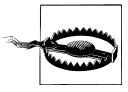
```
(in-package "CLEM")

(defclass matrix ()
  ((m :accessor matrix-vals) ...) ...)

(defmethod (setf mref) (v (m matrix) &rest indices)
  (setf (apply #'aref (matrix-vals m) indices) v))
```

Exercise

Use SLIME to track down these definitions yourself.

Exercise

What does this listing tell you about the relationship between the two packages `CH-IMAGE` and `CLEM`?



If a function calls itself recursively, the time spent in the inner calls will be counted more than once. In some cases a function will appear to be on the stack several hundred percent of the time. This caveat does not just apply to the Metering System.

Cute but Inefficient

We decide to see what can be done to speed up calls to one function. Where do we start? The guilty party in a bottleneck is often the algorithm as a whole. But the function we're looking at here appears to be a very straightforward combination of slot accessors (`image-a`, etc.) and calls to `(setf (apply #'aref))`. It's hard to see how one could make this more efficient other than by tinkering with these function calls.

We can tell from inspection of the code that our array accesses involve repeated function calls to `(setf mref)`. And we know from looking at the source for `(setf mref)` that this function is coded without any assumptions about the arrary's *rank* (how many array indices there are): legal Lisp this may be but not the most efficient. We can improve on this by redefining `clem:mref` as the following *macro*:

```
(in-package "CLEM")

(defmacro mref (matrix &rest indices)
  `(aref (matrix-vals ,matrix) ,@indices))
```

Remember to recompile ch-image:set-argb-values!

```
CL-USER> (time (test 1920000))
(TEST 1920000) took 1,808 milliseconds (1.808 seconds) to run
...
CL-USER>
```

This simple change—telling `aref` at *compile time* how many arguments it's going to get —has reduced the loading time of a JPEG image by one third.

Overhead of Method Calls

The next thing we might look at is the overhead of `set-argb-values` being a generic function. Every time it's called, a generic function has to compute the class of each of its specialized arguments and then use these values to select the applicable method(s). Normally this wouldn't have a noticeable impact on runtimes; but we're in a bottleneck and every overhead can be significant. And as we've already established, the two methods on this generic function are exactly the same, so collapsing them into a single function can't hurt.

```
(in-package "CH-IMAGE")

(defun set-argb-values (img row col a r g b)
  (setf (mref (image-a img) row col) a
        ...))
```

```
CL-USER> (time (test 1920000))
(TEST 1920000) took 1,263 milliseconds (1.263 seconds) to run
...
CL-USER>
```

Results: Inclusive Times

We've shaved nearly half the execution time off `ch-image:read-image-file`. Can we do better?

The gains we've made so far were based on the Metering System's report of exclusive times, which is usually the most productive place to start. It's worth taking a look at inclusive times as well and seeing what we can learn about the full route from the REPL all the way to the bottleneck. We repeat the monitoring exercise, this time profiling the new definition of `set-argb-values` and calling `(monitor:report :nested :inclusive)` to get the report.

Ensure that all monitoring counts have been zeroed and that your new definitions are being watched. The simplest way to achieve both of these together is to unmonitor and remonitor everything, as shown here. If all you need to do is zero the counts, call `(monitor:reset-all-monitoring)`.

```
CL-USER> (monitor:unmonitor)
; No value
CL-USER> (dolist (p '(ch-image ch-util clem jpeg)) (monitor:monitor-all p))
NIL
CL-USER> (ch-image:read-image-file "~/chapter-20/test.jpg")
#<CH-IMAGE:ARGB-8888-IMAGE #x36DDE65E>
CL-USER> (monitor:report :nested :inclusive)

                                %                       Total
Function                        Time  Calls    Sec/Call   Time
-------------------------------------------------------------------
CH-IMAGE:READ-JPEG-FILE:        0.17      1   64.460520  64.460520
CH-IMAGE:READ-IMAGE-FILE:       0.17      1   64.460510  64.460510
CH-IMAGE:JPEG-RGB-TO-ARGB-IMAGE: 0.17     1   63.538517  63.538517
CH-IMAGE:MAP-PIXELS:            0.17      1   63.481556  63.481556
CH-IMAGE:SET-PIXEL:             0.15  1920000   0.000031  58.770000
CH-IMAGE:SET-ARGB-VALUES:       0.14  1920000   0.000027  52.369000
CLEM::MATRIX-VALS:              0.02  7680008   0.000001   8.115990
-------------------------------------------------------------------
TOTAL:                          0.98 11520012             375.196080
...
CL-USER>
```

The fractional times given here have been scaled to add up to 1.0. This was very useful for exclusive times but means absolutely nothing here. Divide by 0.17! Ignore the minor but irrelevant inaccuracy which has reversed the top two functions (by default `monitor:report` lists functions in descending time order).

We can interpret this table as follows:

- The call path is `read-image-file` → `read-jpeg-file` → `jpeg-rgb-to-argb-image` → `map-pixels` → `set-pixel` → `set-argb-values` → `matrix-vals`.
- Almost no time is spent in the first four of these functions.
- A look in the source reveals that `read-jpeg-file` passes the filename to `jpeg:decode-image` and sends the result of that to `jpeg-rgb-to-argb-image`. So it looks like the times spent reading bytes from disk and decoding the JPEG format are insignificant.
- Even with the gains we've already made, over 98% of execution time is spent converting an array of pixel values from one shape into another.

```
(defun jpeg-rgb-to-argb-image (buffer width height)
  (let ((offset 0)
        (img (make-instance 'argb-8888-image :width width :height height)))
    (map-pixels #'(lambda (img x y)
                    (let ((b (svref buffer (ch-util:postincf offset)))
                          (g (svref buffer (ch-util:postincf offset)))
                          (r (svref buffer (ch-util:postincf offset))))
                      (set-pixel img x y (list 255 r g b))))
                img)))
```

Exercise

Locate the source to `ch-util:postincf` and check that it does what you imagine it should. The redefinition below doesn't used this macro; consider whether this is an improvement.

The function `map-pixels` iterates over rows and columns, funcalling its first argument repeatedly; `set-pixel` destructures the list it's been given and calls our old friend `set-argb-values`. Let's see what happens if we rip these out:

```
(defun jpeg-rgb-to-argb-image (buffer width height)
  (let ((offset -1)
        (img (make-instance 'argb-8888-image :width width :height height)))
    (dotimes (row height)
      (dotimes (col width)
        (set-argb-values img row col 255
                         (svref buffer (incf offset))
                         (svref buffer (incf offset))
                         (svref buffer (incf offset)))))
    img))
```

It might not seem like we've done much, but this is our bottleneck and every potential gain is worth investigation. It happens that this change reduces the overall time for `ch-util:read-image-file` by another 10%. Can we do better?

Didn't We Calculate That Two Million Times Before?

Turn back to the inclusive timings table. An eighth of the time is spent in the CLOS reader `clem::matrix-vals`. The call count alone tells us that this must all be inside `read-image-file`. We also know that this function calls four other CLOS readers (`image-a`, `image-r`, etc.). Aha! In all these hundreds of thousands of calls, the results of these readers never vary. See what happens if we call each of these readers just once:

```
(defun jpeg-rgb-to-argb-image (buffer width height)
  (let* ((offset -1)
         (img (make-instance 'argb-8888-image :width width :height height))
         (a-vals (clem::matrix-vals (image-a img)))
         (r-vals (clem::matrix-vals (image-r img)))
         (g-vals (clem::matrix-vals (image-g img)))
         (b-vals (clem::matrix-vals (image-b img))))
    (dotimes (row height)
      (dotimes (col width)
        (setf (aref a-vals row col) 255
              (aref r-vals row col) (svref buffer (incf offset))
              (aref g-vals row col) (svref buffer (incf offset))
              (aref b-vals row col) (svref buffer (incf offset)))))
  img))
```

```
CL-USER> (time (ch-image:read-image-file "~/chapter-20/test.jpg"))
(CH-IMAGE:READ-IMAGE-FILE "~/chapter-20/test.jpg") took 870 milliseconds
...
#<CH-IMAGE:ARGB-8888-IMAGE #x383D9FE6>
CL-USER>
```

We're now down to just over a fifth of our original execution time. It might appear unfortunate that in getting here we bypassed our earlier fixes to `set-argb-values`, but remember that we started out not knowing anything about how this code worked; a couple of false starts aren't that important.

Exercise

Make the ch-image library run faster. Use a suitable level of abstractions, so the code is left looking as tidy as you found it.

The Last Word

Take a look at what the profiler has to say about our latest fix.

```
                                 %                     Total
Function                          Time  Calls  Sec/Call  Time
------------------------------------------------------------
JPEG::DECODE-CHUNK:               0.73      1  1.485998  1.485998
```

```
CLEM::ALLOCATE-MATRIX-VALS:       0.07      4  0.033498  0.133992
JPEG::DECODE:                     0.06 457314  0.000000  0.122372
JPEG::DECODE-FRAME:               0.06      1  0.116998  0.116998
JPEG::INVERSE-COLORSPACE-CONVERT: 0.05      1  0.096998  0.096998
CH-IMAGE:JPEG-RGB-TO-ARGB-IMAGE:  0.04      1  0.085998  0.085998
------------------------------------------------------------------
TOTAL:                            1.00 457416            2.042356
```

The good news is that we've cleared the bottleneck in `jpeg-rgb-to-argb-image`. But before we start popping champagne corks, consider the new "culprit" (`jpeg::decode-chunk`):

- This function was originally on top of the stack 3% of the time.
- We're now running 4 or 5 times faster than we were originally. So we'd expect `decode-chunk` on top of the stack at most 4 or 5 times more often (proportionally) than it was in the first place: no more than 15%.
- These figures are inconsistent.

Exercise

Insert `cl:time` twice into the definition of `ch-image:read-jpeg-file` and verify that the last profiler run was truthful but that the first two reports grossly understated the time spent in `jpeg:decode-image`. Try to explain this.

As already noted, you'll get the best results if you can use a profiler which was written by your Lisp's implementors.

Coding for Speed

Let's put ch-image to one side now and consider some general tips for writing faster code.

If your Lisp's documentation includes chapters on performance, on tuning the compiler, or on efficient use of the GC, read these now.

Data

Before cranking the compiler up to fever pitch it's worth checking that the data structures you're using make sense.

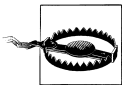
Sequences
: A list gives you flexibility but poor access times other than very near the head of the list (Chapter 9). A vector gives you constant access times down the whole of the sequence but little or no flexibility (Chapter 8).

Dictionaries
: Dictionary lookup among more than a few elements should use a hash table; if the set is very small then either an association list or a property list will do (in terms of performance it doesn't matter which). The break-point will vary between implementations and might depend on the `hash-table-test`; if you can't find anything documented then run your own timing tests. Two points in favor of hash tables: they're more straightforward to code with, and your Lisp ought to provide locking mechanisms for tables which would be harder to implement for lists.

  By default hash tables start off small and *grow* on demand (but not by much) as they fill up. The precise numbers are implementation-specific. As a typical example, tables in Clozure CL start off with space for 60 entries and grow by 50% when they are 85% full; it would take 25 *rehashes* to grow such a table large enough to hold a million entries and each time this happened the location of every entry in the table would have to be recalculated. So if building large hash tables is slowing you down, make sure the tables are created large enough to start with:

```
(make-hash-table :size 1000000)
```

Slot access
: We've already seen that CLOS accessors in vast numbers can cause performance problems. If you don't need *any* of the flexibility of CLOS classes then their somewhat primitive predecessors, the *structures* which have part of Common Lisp since CLtL1, may suffice. Look up the macro `defstruct` to see how these are defined.



Don't use structures unless you really have to, and for preference don't use them at all while your application is under active development. Structure accessors are *inlined*: blinding efficiency which comes at a price. Every time you redefine a structure you'll have to force-recompile every reference to it, quite possibly in a fresh Lisp image.

As a Last Resort

Should all else fail, cranking up the optimization settings might help. Under the right circumstances this can be very effective. Its effects are likely to include:

- inlining of common operations (`car` and `cdr`, arithmetic, array accessors, ...);
- inlining—on request—of your own function calls;
- optimized arithmetic operations on elements of specialized arrays;
- removal of safety mechanisms such as checks on types, array bounds, argument counts, global variables being `boundp`, and so forth;

- consequent low-level errors (segmentation faults rather than the familiar and friendly `"Hey, NIL is not a number so I can't divide by it."`)
- reduced level of debugging information available at run time.

Some compilers (ACL, CMUCL and LW, for instance) are able to *explain* why they were or weren't able to put optimizations into effect. Read the manual!

Optimization is controlled by *declarations*. We've already met one of these (`dynamic-extent` in Chapter 16) along with two other declarations which are nothing to do with optimization (`ignore` and `special` in Chapter 11). We'll discuss three others here; for the full list provided by ANSI CL and further details of the following, see:

http://www.lispworks.com/documentation/HyperSpec/Body/03_cc.htm

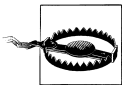
Inline

This declaration permits the compiler to substitute the body of a called function in the place of the call, thus saving the overhead of the function call. The semantics of the called function do not change; the parameters and `return-from` work as they would without the declaration. Note that `inline` has to be in effect both when the inlined function is defined and at the call site (which might be in a different file).

```
(declaim (inline fast-list-position))

(defun fast-list-position (thing list)
  (let ((count 0))
    (loop (let ((next (pop list)))
            (cond ((eq next thing) (return count))
                  ((null list) (return)))
            (incf count)))))

(defun collect-positions (object lists)
  (loop for list in lists collect
       (fast-list-position object list)))
```

would be equivalent to

```
(defun collect-positions (object lists)
  (loop for list in lists collect
       (let ((count 0))
         ;; etc
         ...)))
```



Calls to an inlined function can't be traced. (Or monitored.)

Type

This allows you to make claims about the values of given variables. For example:

```
(declare (type fixnum count index)
         (type simple-string input)
         (type (simple-array (unsigned-byte 8) (*)) result))
```

The symbol `type` in this declaration is optional.

An alternative to `type` declarations is the special-form `the`:

```
(make-array (the fixnum (* size 2)))
```

If the type assertion fails in interpreted code, the implementation is encouraged to tell you nicely. In compiled code all bets are off: you might get an useful error, an uninformative one, or plain rubbish.

```
CL-USER> ((lambda (x) (1+ (the fixnum x))) t)
#<BOGUS object @ #x13012>
CL-USER> ((lambda (x) (car (the cons x))) t)
25165900
CL-USER> (handler-case
            ((lambda (x) (car (the cons x))) 0)
          (serious-condition (e) (princ e)))
Fault during read of memory address #x3
#<CCL::INVALID-MEMORY-ACCESS #x34E687EE>
CL-USER>
```

Exercise

Why didn't I just go `(1+ (the fixnum t))`? (Try it.) What's the point of the `handler-case` in the third example?

Type declarations are a promise that the programmer is making to the compiler. Some implementations, such as CMUCL and SBCL, generally treat declarations as *assertions* (see `cl:assert`).

Optimize

This declaration gives hints to the compiler about how fast, large, safe, etc. you'd like its output to be. There are various *qualities* which takes values from 0 (meaning "not at all") to 3 ("very much, please"). The two really useful settings—please use them with care—tell compiler that you'd like fast, unsafe code:

```
(declare (optimize (speed 3) (safety 0)))
```

If that fails to do the trick, read your documentation to see what else is on offer. In particular, there may be features which allow you to work efficiently with numerical data. For example, LispWorks has a `float` optimization which discourages the compiler from *boxing* intermediate results as valid Lisp objects. The following function allocates 16 bytes (one float object for the final result); without the `float` declaration it allocates twelve times that (one float for each `aref`, `*` and `incf`).

```
(defun dot-product (this that)
  (declare (optimize (safety 0) (float 0))
           ((simple-array double-float (3)) this that))
  (let ((sum 0d0))
    (declare (double-float sum))
```

```
(dotimes (i 3)
  (incf sum (* (aref this i) (aref that i))))
sum))
```

Other numerical optimizations might support fixnum-only arithmetic. You know where to look.

Finally, a brief word about how to use all this. A declaration can be wrapped in `(declare ...)` at the head of a binding form, as introduced in in Chapter 11 and demonstrated in `dot-product` above; in this case its effect is *local* to the form in which it appears.

Alternatively—and this is particularly useful for pervasive speed and safety settings—you can make a declaration *global* by wrapping it in the macro `declaim` at the top level of your source file. If you want to generate the optimizations on the fly (a dynamically controlled safety level, perhaps), use the function `proclaim`:

```
(eval-when (:execute :compile-toplevel :load-toplevel)
  (proclaim `(safety ,(if *safe-mode* 3 0))))
```

It's not always clear how the side-effects of `declaim` persist after the file has been compiled or loaded. For example, see:

http://clozure.com/pipermail/openmcl-devel/2009-August/010168.html

Exercise

Why might we need `eval-when`?

Exercise

Take a look at the source for `jpeg:decode-image` and its callees. Verify that the inner loops are properly optimized.